Guía del programador de NQC

Versión 2.5 a4

por Dave Baum

Índice

1. Introducción

NQC significa Not Quite C (No Completamente C), y es un lenguaje sencillo para programar varios productos LEGO MINDSTORMS. Algunas de las características de NQC dependen del producto MINDSTORMS que se utilice. NQC se refiere a los diferentes ladrillos inteligentes como el *destino*. Actualmente NQC soporta cuatro diferentes destinos: RCX, RCX2 (un RCX que ejecuta el firmware versión 2.0), CyberMaster, Scout y Spybotics.

Todos los destinos tienen un interprete de código de bytes (proporcionado por LEGO) que puede ser utilizado para ejecutar programas. El compilador NQC convierte un programa fuente en LEGO bytecode, el cuál puede ser ejecutado en el propio destino. Aunque las estructuras de preprocesamiento y de control son muy similares a C, NQC no es un lenguaje de propósito general –hay muchas restricciones que son producto de las limitaciones del interprete de código de bytes LEGO.

Lógicamente el NQC se define en dos partes diferenciadas. El lenguaje NQC describe la sintaxis que a utilizar al escribir programas. El NQC API describe las funciones del sistema, constantes y macros que se pueden usar en los programas. Este API se define en un archivo especial incluido en el compilador. Por defecto, este archivo siempre se procesa antes de compilar el programa.

Este documento describe ambos: el lenguaje NQC y el NQC API. En resumen, proporciona la información necesaria para escribir programas en NQC. Dado que hay varios interfaces para NQC, este documento no describe cómo usar ninguna implementación de NQC específica. Consulte la documentación proporcionada con la herramienta NQC, así como el *Manual de Usuario de NQC* para información específica para esa implementación.

Si desea información y documentación actualizada de NQC visite el Sitio Web de NQC en la siguiente dirección:

http://www.baumfamily.org/nqc

2. Lenguaje NQC

Esta sección describe el propio lenguaje NQC. Incluye las reglas léxicas usadas por el compilador, la estructura de los programas, sentencias y expresiones y el funcionamiento del preprocesador.

2.1. Reglas léxicas

Las reglas léxicas describen cómo NQC divide un archivo fuente en tokens individuales. Incluye cómo están escritos los comentarios, el manejo de los espacios en blanco y los caracteres válidos para identificadores.

2.1.1. Comentarios

NQC soporta dos tipos de comentarios. El primer tipo (comentarios C tradicionales) empieza con /* y termina con */. Pueden abarcar varias líneas, pero no pueden anidarse.

```
/* esto es un comentario */

/* esto es un comentario
de dos líneas */

/* otro comentario…
    /* intentando anidar…
    finalizando el comentario interior… */
este texto ya no es un comentario!  */
```

La segunda clase de comentarios empieza con // y termina con una nueva línea (conocidos a veces como comentarios estilo C++).

```
// un comentario de una línea sencilla
```

El compilador ignora los comentarios. Su único propósito es permitir al programador documentar el código fuente.

2.1.2. Espacio en blanco

El espacio en blanco (espacios, tabuladores y saltos de línea) se usa para separar tokens y para hacer los programas más legibles. Con tal que se distingan los tokens, no tiene efecto en el programa el añadir o suprimir espacios en blanco. Por ejemplo, las siguientes líneas de código tienen el mismo significado:

```
x=2;

x = 2 ;
```

Algunos de los operadores de C++ están formados por múltiples caracteres. Para preservar estos tokens no se debe insertar espacio en blanco entre ellos. En el ejemplo siguiente, la primera línea usa un operador de cambio a la derecha (“>>”), pero en la segunda línea el espacio añadido hace que los símbolos “>” se interpreten como dos elementos separados y en consecuencia se genere un error.

```
x = 1 >> 4;  //dar a x el valor de 1 desplazado a la derecha 4 bits

x = 1 > > 4; // error
```

2.1.3. Constantes numéricas

Las constantes numéricas se pueden escribir tanto en formato decimal como hexadecimal. Las constantes decimales consisten en uno o más dígitos decimales. Las constantes hexadecimales empiezan con 0x ó 0X seguidos de uno o más dígitos hexadecimales.

```
x = 10;                 // dar a x el valor 10
x = 0x10;         // dar a x el valor 16 (10 hex)
```

2.1.4. Identificadores y palabras reservadas

Se usan los identificadores para nombres de tarea, variables y funciones. El primer carácter de un identificador debe ser una letra mayúscula o minúscula o el guión bajo (“_”). Los caracteres restantes pueden ser letras, números y el guión bajo.

Se reserva un número potencial de identificadores para uso del propio lenguaje NQC. Estas palabras se denominan palabras reservadas y no se pueden usar como identificadores. A continuación se ofrece la lista completa de palabras reservadas:

`__event_src`	`__res`	`__taskid`	`abs`
`__nolist`	`__sensor`	`__type`	`acquire`
`asm`	`do`	`int`	`sub`
`break`	`else`	`monitor`	`switch`
`case`	`false`	`repeat`	`task`
`catch`	`for`	`return`	`true`
`const`	`goto`	`sign`	`void`
`continue`	`if`	`start`	`while`
`default`	`inline`	`stop`	

2.2. Estructura del programa

Un programa NQC se compone de bloques de código y variables globales. Hay tres tipos de bloques de código: tareas, funciones en línea y subrutinas. Cada tipo de bloque de código tiene sus propias características y restricciones particulares, pero todos comparten una estructura común.

2.2.1. Tareas

RCX soporta implícitamente multitarea, de modo que una tarea en NQC corresponde a una tarea RCX. Las tareas se definen por medio de la palabra reservada `task` utilizando la siguiente sintaxis:

```
task nombre()
{
   // el código de la tarea se escribe aquí
}
```

El nombre de la tarea puede ser cualquier identificador legal. Un programa debe tener al menos una tarea llamada main –que se inicia cada vez que se ejecuta el programa. El número máximo de tareas depende del destino –RCX soporta 10 tareas, CyberMaster 4 y Scout 6.

El cuerpo de una tarea consiste en una lista de instrucciones. Las tareas pueden iniciarse y detenerse utilizando las instrucciones `start` y `stop` (descritas en la sección titulada *Instrucciones*). Hay también un comando RCX API, `StopAllTasks`, que detiene todas las tareas en funcionamiento en ese momento.

2.2.2. Funciones

A menudo es útil agrupar un conjunto de instrucciones en una sola función, que puede ser llamada cuando sea necesario. NQC soporta funciones con argumentos, pero no valores de retorno. Las funciones se definen con la siguiente sintaxis:

```
void nombre(lista_de_argumentos)
{
    // cuerpo de la función
}
```

La palabra reservada `void` es consecuencia de la herencia de NQC –en C las funciones se especifican con el tipo de datos que devuelven. Las funciones que no devuelven datos devuelven `void`. La devolución de datos no está soportada por NQC, de modo que todas las funciones se declaran usando la palabra reservada `void`.

La lista de argumentos puede estar vacía o puede contener una o más definiciones de argumento. Un argumento se define por su tipo seguido de su nombre. Los argumentos múltiples se separan por comas. Todos los valores en RCX se representan como enteros de 16 bits con signo. Sin embargo, NQC soporta cuatro tipos diferentes de argumentos que corresponden a diferentes restricciones y modos de paso del argumento:

Tipo	Significado	Restricción
Int	Paso por valor	ninguna
const int	Paso por valor	sólo se pueden usar constantes
int&	Paso por referencia	sólo se pueden usar variables
const int	Paso por referencia	la función no puede modificar el argumento

Los argumentos del tipo `int` se pasan por valor desde la función que llama a la función llamada. Esto quiere decir que el compilador debe asignar una variable temporal que almacene el argumento. No hay restricciones en el tipo de valor que puede usarse. Sin embargo, ya que la función trabaja con una copia del argumento en sí, cualquier cambio que sufra el valor no lo ve la función que la llama. En el ejemplo de abajo la función `foo` intenta establecer el valor de su argumento en 2. Esto es perfectamente legal, pero dado que `foo` funciona en una copia del argumento original, la variable `y` de la tarea principal no sufrirá cambios.

```
void foo(int x)
{
    x = 2;
}
task main()
{
```

```
    int y = 1;          // y es ahora igual a 1
    foo (y);      // y es todavía igual a 1!
}
```

El segundo tipo de argumento, `const int`, también se pasa por valor, pero con la restricción de que sólo se pueden usar valores constantes (es decir, números). Esto es muy importante ya que hay varias funciones RCX que sólo funcionan con argumentos constantes.

```
void foo(const int x)
{
    PlaySound (x);      // ok
    x = 1;        // error – no puede modificar un argumento
}

task main()
{
    foo (2);            // ok
    foo (4*5 );   // ok – expresión todavía constante
    foo (x);            // error - x no es una constante
}
```

El tercer tipo, `int &`, pasa argumentos por referencia en vez de por valor. Esto permite que la función llamada modifique el valor y haga visibles los cambios en la función que llama. Sin embargo, sólo se pueden usar variables cuando se llama a una función usando argumentos `int &`:

```
void foo(int &x)
{
    x = 2;
}

task main()
{
    int y = 1;          // y es igual a 1
    foo (y);            // y es ahora igual a 2
    foo (2);            // error – solo se permiten variables
}
```

El último tipo, `const int &`, es muy poco frecuente. También se pasa por referencia, pero con la restricción de que a la función llamada no se le permite modificar el valor. A causa de esta restricción, el compilador puede pasar cualquier cosa (no sólo variables) a funciones usando este tipo de argumento. En general esta es la forma más eficiente de pasar argumentos en NQC.

Hay una diferencia importante entre argumentos `int` y argumentos `const int &`. Un argumento `int` se pasa por valor, de modo que en el caso de una expresión

dinámica (como la lectura de un sensor), el valor se lee una vez y se almacena. Con los argumentos `const int &`, la expresión se lee cada vez que es usada en la función:

```
void foo(int x)
{
    if (x == x)                // esto siempre es verdad
    PlaySound(SOUND_CLICK);
}

void bar(const int &x)
{
    if (x == x)                //puede no ser verdad..
                                     // el valor podría cambiar
    PlaySound(SOUND_CLICK);
}

task main()
{
    foo(SENSOR_1);             // reproduce sonido
    bar(2);                    // reproduce sonido
    bar(SENSOR_1);             // podría no reproducir sonido
}
```

Las funciones deben invocarse con el número (y tipo) correcto de argumentos. El ejemplo siguiente muestra diferentes llamadas legales e ilegales a la función foo:

```
void foo(int bar, const int baz)
{
    // haz algo aquí...
}

task main()
{
    int x;         // declarar variable x
    foo (1,2);           // ok
    foo (x,2);           // ok
    foo (2,x);           // error – segundo argumento no constante!
    foo (2);             // error – numero equivocado de argumentos!
}
```

Las funciones NQC siempre se expanden como funciones en línea. Esto significa que cada llamada a una función hace que se incluya en el programa otra copia del código de la función. Si no se usan con sensatez, las funciones en línea hacen que el tamaño del código sea excesivo.

2.2.3. Subrutinas

A diferencia de las funciones en línea las subrutinas permiten que se comparta una única copia de un fragmento código entre diferentes funciones llamadoras. Esto hace que sean mucho más eficaces en el uso del espacio que las funciones en línea, pero debido a algunas limitaciones en el interprete de código de bytes LEGO, las subrutinas tiene algunas restricciones significativas. En primer lugar, las subrutinas no pueden utilizar ningún argumento. Segundo, una subrutina no puede llamar a otra subrutina. Por último, el máximo número de subrutinas se limita a 8 para el RCX, 4 para CyberMaster, 3 para Scout y 32 para el Spybotics. Además, cuando se utiliza el RCX 1.0 o el CyberMaster, si una subrutina es llamada desde múltiples tareas no puede tener variables locales o realizar cálculos que requieran variables temporales. Estas importantes restricciones hacen que las subrutinas sean menos atractivas que las funciones, por lo tanto su uso debería limitarse a situaciones donde sea absolutamente necesario ahorrar en el tamaño del código. A continuación se ofrece la sintaxis de una subrutina:

```
Nombre_de_la_subrutina()
{
	// cuerpo de la subrutina
}
```

2.2.4. Variables

Todas las variables en NQC son del mismo tipo –enteros con signo de 16 bits. Las variables se declaran usando la palabra reservada `int` seguida de una lista de nombres de variables separados por coma y terminados por un punto y coma (“;”). Opcionalmente, se puede especificar un valor inicial para cada variable usando el signo (“=”) después del nombre de la variable. Veamos a continuación varios ejemplos:

```
int x;			// declara x
int t,z;		// declara t y z
int a=1,b;	// declara a y b, inicializa a con el valor 1
```

Las variables globales se declaran en el ámbito del programa (fuera de cualquier bloque de código). Una vez declaradas se pueden utilizar dentro de cualquier tarea, función y subrutina. Su ámbito comienza en la declaración y termina al final del programa.

Las variables locales se pueden declarar dentro de las tareas, funciones y a veces dentro de las subrutinas. Tales variables sólo son accesibles dentro del bloque del código en el que se definen. Concretamente, su ámbito empieza con la declaración y termina al final del bloque de código. En el caso de las variables locales se considera un bloque una instrucción compuesta (un grupo de instrucciones incluidas entre dos llaves “{“ y “}”):

```
int x; // x es global


task main()
{
	int y;		// y es local de la tarea main
	x = y;		// ok
```

```
    {                  // comienza la instrucción compuesta
    int z;      // z local declarada
          y = z;      // ok
  }
    y = z;            // error – aquí no está definida z
}

task foo ()
{
    x = 1;            // ok
    y = 2;            // error - y no es global
}
```

En muchos casos NQC debe asignar una o más variables temporales para su propio uso. En algunos casos se utiliza una variable temporal para alojar un valor intermedio durante un cálculo. En otros casos se utiliza para guardar un valor al ser pasado a función. Estas variables temporales reducen la reserva de variables disponibles para el resto del programa. NQC intenta ser lo más eficiente posible con las variables temporales (incluso reutilizándolas siempre que sea posible).

El RCX (y otros destinos) proporcionan varias posiciones de almacenamiento que se pueden usar para alojar variables en un programa NQC. Hay dos tipos de posiciones de almacenamiento: globales y locales. Cuando compila un programa, NQC asigna cada variable a un lugar específico de almacenamiento. Los programadores generalmente pueden ignorar los detalles de esta asignación siguiendo dos reglas básicas:

- Si una variable necesita estar en posición global se declara como variable global.
- Si una variable no necesita ser global, hacer que sea lo más local posible.

Esto da al compilador la máxima flexibilidad al asignar una posición de almacenamiento concreta.

El número de variables locales y globales varía según el destino:

Destino	Global	Local
RCX	32	0
CyberMaster	32	0
Scout	10	8
RCX2	32	16
Spybotics	32	4

2.2.5. Matrices

Los destinos RCX2 y Spybotics soportan matrices (los otros destinos no tienen soporte apropiado para matrices en el firmware). Las matrices se declaran de la misma forma que las variables normales, pero con el tamaño de la matriz encerrado entre corchetes. El tamaño debe ser una constante.

```
int mi_matriz[3]; // declara una matriz de tres elementos
```

Los elementos de una matriz se identifican por su posición dentro de la matriz (llamada índice). El primer elemento tiene un índice 0, el segundo 1, etc. Por ejemplo:

```
mi_matriz[0]=123;            // establece el primer elemento en 123
mi_matriz[1]= mi_matriz[2];// copia el tercero en el segundo
```

Actualmente hay una serie de limitaciones en el uso de matrices. Es probable que dichas limitaciones se supriman en las futuras versiones de NQC:

- Una matriz no puede ser el argumento de una función. Sin embargo, se puede pasar a una función un elemento individual de la matriz.
- Ni las matrices si sus elementos pueden utilizarse con los operadores de aumento (++) o disminución (--).
- Sólo se permite la asignación normal (=) para los elementos de la matriz. No se permiten las asignaciones matemáticas (+=).
- Los valores iniciales de los elementos de una matriz no se pueden especificar. Se requiere que una asignación explícita dentro del mismo programa que establezca el valor de un elemento.

2.3. Instrucciones

El cuerpo de un bloque de código (tarea, función o subrutina) se compone de instrucciones. Las instrucciones se terminan con un punto y coma (“;”).

2.3.1. Declaración de variables

La declaración de variables, como se ha descrito en la sección anterior, es un tipo de instrucción. Una variable se declara como local (con inicialización opcional) cuando ha de ser utilizada dentro de un bloque de código. La sintaxis para una declaración de variable es:

```
int variables;
```

donde variables es una lista de nombres separados por comas con valores iniciales opcionales.

```
nombre [=expresión]
```

Las matrices de variables también se pueden declarar (sólo para RCX2):

```
int matriz[tamaño];
```

2.3.2. Asignación

Una vez declaradas las variables se les puede asignar el valor de una expresión:

```
Variable operador_de_asignación expresión;
```

Hay nueve operadores de asignación. El operador más básico, “=”, simplemente asigna el valor de la expresión a la variable. Los otros operadores modifican de alguna forma el valor de la variable en los modos en que se muestra en la tabla siguiente.

Operador	Acción
=	Asigna a una variable una expresión
+=	Añade a una variable una expresión
-=	Resta a una variable una expresión
*=	Multiplica a una variable por una expresión
/=	Divide una variable por una expresión
&=	AND bit a bit de la expresión y la variable
|=	OR bit a bit de la expresión y la variable
||=	Asigna a una variable el valor absoluto de una expresión
+-=	Asigna una variable el signo (-1, +1, 0) de una expresión
>>=	Desplaza a al derecha la variable en una cantidad constante
<<=	Desplaza a al izquierda la variable en una cantidad constante

Algunos ejemplos:

```
x = 2;          // asigna a x el valor 2
y = 7;          // asigna a y el valor 7
x += y;         // x es 9, y es todavía 7
```

2.3.3. Estructuras de control

La estructura de control más sencilla es una instrucción compuesta. Esto es una lista de instrucciones encerradas entre llaves ("{" y "}").

```
{
   x = 1;
   y = 2;
}
```

Aunque puede no parecer muy significativo, juega un papel crucial al construir estructuras de control más complicadas. Muchas estructuras de control requieren una instrucción sencilla como cuerpo. Usando una instrucción compuesta, la misma estructura de control se puede usar para controlar múltiples instrucciones.

La instrucción `if` evalúa una condición. Si la condición es verdadera ejecuta una instrucción (la consecuencia). Una segunda instrucción opcional (la alternativa) se ejecuta si la condición es falsa. A continuación se muestran las dos sintaxis posibles para una instrucción `if`.

```
if (condición) consecuencia
if (condición) consecuencia else alternativa
```

Obsérvese que la condición va encerrada entre paréntesis. Véanse los ejemplos a continuación. En el último ejemplo se utiliza una instrucción compuesta para permitir que se ejecuten dos instrucciones como consecuencia de la condición.

```
if (x==1) y = 2;
if (x==1) y = 3; else y = 4;
if (x==1) {y = 1; z = 2;}
```

La instrucción `while` se usa para construir un bucle condicional. La condición se evalúa y si es verdadera se ejecuta el cuerpo del bucle, a continuación se comprueba de nuevo la condición. El proceso continúa hasta que la condición se vuelve falsa (o se ejecuta la instrucción `break`). A continuación aparece la sintaxis para el bucle `while`:

```
while (condición) cuerpo
```

Es normal utilizar una instrucción compuesta como cuerpo del bucle.

```
while (x < 10)
{
    x = x+1;
    y = y*2;
}
```

Una variante del bucle `while` es el bucle `do-while`. Su sintaxis es:

```
do cuerpo while (condición)
```

La diferencia entre el bucle `while` y el `do-while` es que el `do-while` siempre ejecuta el cuerpo al menos una vez mientras que el bucle `while` puede no ejecutarlo nunca.

Otro tipo de bucle es el bucle `for`:

```
for (instr1 ; condición ; instr2 ;) cuerpo
```

Un bucle `for` siempre ejecuta instr1, luego chequea repetidamente la condición y mientras es verdadera ejecuta el cuerpo seguido de instr2. El bucle `for` es equivalente a:

```
instr1;
while (condición)
{
    cuerpo
    instr2;
}
```

La instrucción `repeat` ejecuta un bucle un número determinado de veces:

```
repeat (expresión) cuerpo
```

La expresión determina cuantas veces se ejecutará el cuerpo. Obsérvese la expresión sólo se evalúa una vez y el cuerpo se repite ese número de veces. Esto es diferente en las estructuras `while` y `do-while` que evalúan la condición en cada bucle.

Una instrucción `switch` se puede utilizar para ejecutar uno de varios bloques de código dependiendo del valor de una expresión. Cada bloque de código va precedido por una o más etiquetas `case`. Cada `case` debe ser una constante única dentro de la instrucción `switch`. La instrucción `switch` evalúa la expresión y a continuación busca una etiqueta `case` que cumpla la condición. Entonces ejecutará cualquier instrucción que siga a dicho `case` hasta que se encuentre una instrucción `break` o hasta que llegue al final del `switch`. También se puede usar una única etiqueta `default` que se asociará a cualquier valor que no aparezca en la etiqueta `case`. Técnicamente una instrucción `switch` tiene la siguiente sintaxis:

```
switch (expresion) cuerpo
```

Las etiquetas `case` y `default` no son instrucciones en sí mismas sino que son *etiquetas* que preceden a las instrucciones. Múltiples etiquetas pueden preceder a la misma instrucción. Estas etiquetas tienen la siguiente sintaxis:

```
case expresión_constante:
default :
```

Una típica instrucción `switch` tendría este aspecto:

```
switch (x)
{
   case 1 :       // haz algo cuando x es 1
          break;
   case 2 :
   case 3 :       // haz otra cosa cuando x es 2 ó 3
          break;
   default :           // haz esto cuando x no es 1,2 ni 3
   break;
}
```

La instrución `goto` fuerza al programa a saltar a una posición determinada. Las instrucciones de un programa pueden ser marcadas precediendolas de un identificador y dos puntos. Una instrucción goto especifica la etiqueta a la que el programa ha de saltar. Por ejemplo, veamos cóm,o implementar un bucle que incrementa el valor de una variable utilizando `goto`:

```
mi_bucle:
x++;
goto mi_bucle;
```

La instrucción goto debe usarse con moderación y precaución. En la mayoría de los casos las estructuras de control tales como if , while y switch hacen los programas más fáciles de leer y modificar. Ten la precaución de no utilizar nunca una instrucción goto para saltar a una instrucción `monitor` o `acquire` o para salir de ella. Esto es así porque `monitor` y `acquire` tienen un código especial que normalmente es ejecutado en la entrada y salida, y un goto evitaría este código – probablemente provocando un comportamiento no deseado.

NQC también define el macro `until` que supone una alternativa práctica al bucle while. La definición real de until es:

```
#define until (c ) while (!( c ))
```

En otras palabras, `until` continuará haciendo bucles hasta que la condición sea verdadera. Con mucha frecuencia se utiliza con una instrucción de cuerpo vacío:

```
until (SENSOR_1 = 1); // espera a que se presione el sensor
```

2.3.4. Control de acceso y eventos

El Scout, RCX2 y Spybotics soportan la monitorización de eventos y el control de acceso. El control de acceso permite que una tarea solicite la posesión de uno o más recursos. En NQC el control de acceso lo proporciona la instrucción `acquire`, que puede presentar dos formas:

```
acquire (recursos) cuerpo
acquire (recursos) cuerpo catch handler
```

donde recursos es una constante que especifica los recursos que hay que obtener y cuerpo y handler son instrucciones. El NQC API define las constantes para los recursos individuales que se pueden sumar para solicitar múltiples recursos a la vez. El comportamiento de la instrucción `acquire` es el siguiente: solicitará la posesión de los recursos especificados. Si otra tarea de prioridad superior ya posee los recursos, entonces la solicitud no será aceptada y la ejecución saltará al handler (si existe). En caso contrario la petición será aceptada y el cuerpo empezará a ejecutarse. Mientras se ejecuta el cuerpo, si una tarea de prioridad igual o superior solicita alguno de los recursos la tarea original perderá su posesión. Cuando se pierde la posesión, la ejecución salta al handler (si existe). Una vez que el cuerpo se ha completado se devuelven los recursos al sistema (para que puedan adquirirlos tareas de prioridad más baja). Si no se especifica un handler, tanto si la petición no es aceptada como si lo es (con su consecuente pérdida una vez ejecutado el cuerpo) el control pasa a la instrucción que sigue a la instrucción `acquire`. Por ejemplo el siguiente código adquiere un recurso 10 segundos, haciendo sonar un sonido si no se completa con éxito:

```
acquire (ACQUIRE_OUT_A)
{
   Wait (1000);
}

catch
{
   PlaySound (SOUND_UP);
}
```

La monitorización de eventos se implementa con la instrucción `monitor`, que tiene una sintaxis muy similar a la instrucción `acquire`:

```
monitor (eventos) cuerpo
monitor (eventos) cuerpo handler_list
```

donde handler_list es uno o más handlers del tipo:

```
catch (catch_events) handler
```

El último handler en una lista de handlers puede omitir la especificación del evento:

```
catch handler
```

Evento es una constante que determina que eventos deben monitorizarse. Para el Scout los eventos están predefinidos, de modo que hay constantes como `EVENT_1_PRESSED` que pueden ser utilizar como especificación de evento. Con RCX2 el significado de cada evento es configurado por el programador. Hay 16 eventos

(números del 0 al 15). Para especificar un evento en una instrucción de monitor, el número de evento debe convertirse en una máscara de evento utilizando la macro `EVENT_MASK()`. Las constantes de evento del Scout o máscaras de evento pueden sumarse para especificar múltiples eventos. Pueden combinarse múltiples máscaras por medio del OR bit a bit.

La instrucción `monitor` ejecuta el cuerpo mientras monitoriza los eventos especificados. Si sucede cualquiera de los eventos, la ejecución salta al primer handler para ese evento (un handler sin ninguna especificación maneja cualquier evento). Si no existe ningún handler de evento para ese evento, entonces el control continúa en la instrucción que sigue a la instrucción `monitor`. El siguiente ejemplo espera 10 segundos mientras monitoriza los eventos 2, 3 y 4 para RCX2:

```
monitor (EVENT_MASK (2)| EVENT_MASK (3)| EVENT_MASK (4))
{
   Wait (1000);
}
catch (EVENT_MASK (4))
{
   PlaySound (SOUND_DOWN); // sucedió el evento 4
}
catch
{
   PlaySound (SOUND_UP); // sucedió el evento 2 ó 3
}
```

Observe que las instrucciones `acquire` y `monitor` sólo son soportadas por destinos que implementen el control de acceso y la monitorización de eventos, es decir, el Scout y el RCX2.

2.3.5. Otras instrucciones

Una llamada a función (o subrutina) es una instrucción como la siguiente:

```
Nombre(argumentos);
```

La lista de argumentos es una lista de expresiones separadas por comas. El número y el tipo de argumentos que se proporcionan debe coincidir con la definición de la función misma.

Las tareas deben iniciarse y terminarse con las siguientes instrucciones:

```
start nombre_de_tarea;
stop nombre_de_tarea;
```

Dentro de los bucles (por ejemplo, en un bucle `while`) la instrucción `break` puede utilizarse para salir del bucle y la instrucción `continue` se puede utilizar para saltar a la parte superior de la siguiente iteración del bucle. La instrucción `break` también puede utilizarse para salir de la instrucción `switch`.

```
break;
continue;
```

Es posible hacer que una función finalice antes de llegar al final de su código usando la instrucción `return`.

```
return;
```

Cualquier expresión es también una instrucción legal cuando termina en punto y coma. Es poco frecuente usar ese tipo de instrucción ya que entonces se descartaría el valor de la expresión. La única excepción mencionable se refiere a las expresiones que implican los operadores de incremento (++) o decremento (`--`).

```
X++;
```

La instrucción vacía (sólo el punto y coma) es también una instrucción legal.

2.4. Expresiones

Las primeras versiones de NQC hacían una distinción entre expresiones y condiciones. Esta diferencia se eliminó a partir de la versión 2.3: todo es una expresión y hay ahora operadores condicionales para las expresiones. Esto es parecido a cómo trata C/C++ las operaciones condicionales.

Los *valores* son el tipo más primitivo de expresiones. Se forman expresiones más complicadas a partir de valores usando varios operadores. El lenguaje NQC sólo tiene incorporado dos clases de valores: constantes numéricas y variables. El RCX API define otros valores que corresponden a varias características del RCX tales como sensores y temporizadores (timers).

Las constantes numéricas en el RCX se representan como enteros con signo de 16 bits. Internamente NQC usa matemáticas con signo de 32 bits para la evaluación de expresiones constantes, luego lo reduce a 16 bits cuando genera el código RCX. Las constantes numéricas se pueden escribir como decimales (`123`) o hexadecimales (`0xABC`). Actualmente, hay muy poco control del rango de valor de las constantes, de modo que usar un valor más grande de lo esperado puede tener efectos inusuales.

Se predefinen dos valores especiales: `true` y `false`. El valor de `false` es cero, mientras que sólo se garantiza que el valor de `true` es no-cero. Son válidos los mismos valores para operadores relacionales (<): cuando la relación es falsa, el valor es cero, en cualquier otro caso el valor es no-cero.

Se pueden combinar los valores utilizando operadores. Varios de los operadores sólo se pueden usar al evaluar expresiones constantes, lo que significa que sus operandos deben ser bien constantes, bien expresiones que no impliquen nada excepto constantes. Se presenta una lista de operadores en orden de prioridad (más alta a más baja).

Operador	Descripción	Asociatividad	Restricción	Ejemplo
abs ( ) sign ( )	Valor absoluto Signo de operando	n/a		abs (x) sign (x)
++, --	Incremento, Disminución	Izquierda	Sólo variables	x++ o ++x
- ~ !	Menos unario Negación bitwise (unario) Negación lógica	Derecha Derecha Derecha	Sólo constantes	-x ~123 !x
*, /, %	Multiplicación, división, modulo	Izquierda		x * y
+, -	Suma, resta	Izquierda		x + y
<<, >>	Desplazamiento a derecha e izquierda	Izquierda	La medida del desplazamiento ha de ser constante	x<<4
<, > <=, >=	Operadores relacionales	Izquierda		x < y
= =, !=	Igual a, no igual a	Izquierda		x = = 1
&	AND bit a bit	Izquierda		x & y
^	XOR bit a bit	Izquierda		x ^ y
		OR bit a bit	Izquierda		x	y
&&	AND lógico	Izquierda		x && y
||	OR lógico	Izquierda		x || y
? :	Valor condicional	n/a		x==1 ? y : z

Se pueden usar paréntesis donde sea necesario para cambiar el orden de evaluación.

```
x = 2+3; 4        // asignar a x el  valor 14
y = (2+3)*4             // asignar a y el valor 20
```

2.4.1. Condiciones

Las condiciones se forman generalmente comparando dos expresiones. Hay también dos expresiones constantes – `true` y `false` – que siempre dan como valor de evaluación verdadero o falso respectivamente. Se puede negar una condición con el operador de negación o combinar dos condiciones con los operadores AND u OR. La tabla siguiente resume los diferentes tipos de condiciones.

Condición	Significado
true	Siempre verdadero
false	Siempre falso
expresión	Verdad si expresión no es igual a 0
expr1 == expr2	Verdad si expr1 es igual a expr2
expr1 != expr2	Verdad si expr1 no es igual a expr2
expr1 < expr2	Verdad si una expr1 es menos que expr2
expr1 <= expr2	Verdad si expr1 es menor o igual a expr2
expr1 > expr2	Verdad si expr1 es mayor que expr2
expr1 >= expr2	Verdad si expr1 es mayor o igual a expr2
! condición	Negación lógica de una condición – verdadero si la condición es falsa
cond1 && cond2	AND lógico de dos condiciones (verdadero si y sólo si ambas condiciones son verdad)
cond1 || cond2	OR lógico de dos condiciones (verdad si y sólo si al menos una es verdad)

2.5. El preprocesador

El preprocesador implementa las siguientes directrices: `#include`, `#define`, `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else`, `#endif`, `#undef`. Su implementación está muy próxima a un preprocesador C estándar, de modo que la mayor parte de las cosas que funcionan en un preprocesador C genérico deberían tener el efecto esperado en NQC. Abajo aparece una lista de desviaciones significativas.

2.5.1. #include

El comando `#include` funciona como era de esperar con la salvedad de que el nombre del archivo debe ir encerrado entre comillas. No hay noción en el sistema de una ruta de inclusión , de modo que encerrar un archivo en paréntesis angular está prohibido.

```
#include “foo.nqh” // ok

#include <foo.nqh> // error!
```

2.5.2. #define

El comando `#define` se usa para una sustitución de macro sencilla. La redefinición de macro es un error (al revés que en C donde es un aviso). Las macros se terminan normalmente por el final de línea, pero puede escribirse la nueva línea con (“\”) para permitir macros multilínea.

```
#define foo (x) do { bar (x);\
                        baz (x); } while (false)
```

La directiva `#undef` se puede usar para retirar una definición de macro.

2.5.3. Compilación condicional

La compilación condicional funciona de forma parecida al preprocesador C. Se pueden usar las siguientes directivas de preprocesador.

```
#if condición

#ifdef symbol

#ifndef symbol

#else

#elif condición

#endif
```

La condiciones en las directivas #if usan los mismos operadores y prioridad que en C. Se soporta el operador defined( ).

2.5.4. Inicialización del programa

Al principio del programa el preprocesador inserta una llamada a una función de inicialización especial, `_init`. Esta función por defecto es parte del RCX API y pone las tres salidas a plena potencia en dirección hacia delante (todavía desconectadas). La función de inicialización se puede desactivar usando la directiva `#pragma noinit`:

```
#pragma noinit          //no hacer ninguna inicialización de programa
```

La función de inicialización por defecto se puede reemplazar por una función diferente usando la directiva `#pragma init`:

```
#pragma init función          //usar inicialización de usuario
```

2.5.5. Almacenamiento de reserva

El compilador NQC asigna automáticamente variables a posiciones de almacenamiento. Sin embargo, a veces es necesario impedir que el compilador use ciertas posiciones de almacenamiento. Esto se puede hacer por medio de la directiva `#pragma reserve`:

```
#pragma reserve inicio

#pragma reserve inicio fin
```

Esta directiva hace que el compilador ignore una o más posiciones de almacenamiento durante la asignación de variables. *Inicio* y *fin* deben ser números que se refieran a posiciones de almacenamiento válidas. Si sólo se proporciona un *inicio*, entonces se reserva una única posición. Si se especifican ambos *inicio* y *fin* entonces se reserva la gama de posiciones de principio a fin (inclusive). El uso más común de esta directiva es para reservar las posiciones 0, 1 y/ó 2 al usar contadores para RCX2. Esto es por que los contadores del RCX2 se solapan con las posiciones de almacenamiento 0, 1 y 2. Por ejemplo, si se fuesen a utilizar los tres contadores:

```
#pragma reserve 0 1 2
```

3. NQC API

El NQC API define un grupo de constantes, funciones, valores y macros que proporcionan acceso a varias capacidades del destino como sensores, salidas, temporizadores y comunicaciones. Algunas características sólo se encuentran en ciertos destinos. Siempre que sea necesario, el título de la sección indica a qué destino se aplica. El RCX2 reúne todas las características del RCX, de modo que si se hace referencia al RCX, entonces esa característica funciona con el firmware original y con el firmware 2.0. Si se hace referencia al RCX2 la característica sólo se aplica al firmware 2.0. CyberMaster, Scout y Spybotics se indican como CM, Scout y Spy respectivamente.

El API consiste en funciones, valores y constantes. Una función es algo que puede ser también denominado instrucción. Típicamente emprende alguna acción o configura algún argumento. Los valores representan algún argumento o cantidad y se pueden usar en expresiones. Las constantes son nombres simbólicos para valores que tienen significado especial para el destino. A menudo se usa un grupo de constantes junto con una función. Por ejemplo la función PlaySound utiliza un sencillo argumento que determina qué sonido se tocará. Las constantes, tales como SOUND_UP, se definen para cada sonido.

3.1. Sensores

Hay tres sensores, que se numeran internamente 0, 1 y 2. Esto podría prestarse a confusión ya que en el RCX están etiquetados externamente como 1, 2 y 3. Para mitigar esta confusión se han definido los nombres de sensor SENSOR_1, SENSOR_2 y SENSOR_3. Estos nombres de sensor se pueden usar en cualquier función que requiera un sensor como argumento. Además, se pueden usar los nombres siempre que un programa desee leer el valor actual de un sensor:

```
x = SENSOR_1;          // leer el sensor y almacenar el valor en x
```

3.1.1. Tipos y modos — RCX, CyberMaster

Los puertos de sensor en el RCX tienen capacidad de soportar una gran variedad de sensores (otros destinos no soportan tipos de sensor configurables). Es función del programa decirle al RCX qué clase de sensor está conectado en cada puerto. Se puede configurar el tipo de sensor por medio de `SetSensorType`. Hay cuatro tipos de sensores y cada cual corresponde a un sensor específico de LEGO. Se puede usar un quinto tipo (SENSOR_TYPE_NONE) para leer los valores puros de los sensores genéricos pasivos. En general, un programa debería configurar el tipo para que encaje con el sensor real. Si un puerto de sensor se configura con el tipo incorrecto, el RCX puede no ser capaz de leerlo correctamente.

Tipo de sensor	Significado
SENSOR_TYPE_NONE	Sensor pasivo genérico
SENSOR_TYPE_TOUCH	Sensor de tacto
SENSOR_TYPE_TEMPERATURE	Sensor de temperatura
SENSOR_TYPE_LIGHT	Sensor de luz
SENSOR_TYPE_ROTATION	Sensor de rotación

El RCX, CyberMaster y Spybotics permiten que se configure un sensor en modos diferentes. El modo de sensor determina cómo se procesa el valor puro de un sensor. Algunos modos sólo tienen sentido para cierto tipo de sensores, por ejemplo SENSOR_MODE_ROTATION sólo es útil con sensores de rotación. El modo de sensor se puede establecer por medio de `SetSensorMode`. A continuación se muestran los posibles modos. Adviértase que, ya que CyberMaster no soporta sensores de rotación o de temperatura, los últimos tres modos se restringen sólo al RCX. Spybotics es todavía más restrictivo y solo permite los modos raw, booleano y porcentual.

Modo de sensor	**Significado**
SENSOR_MODE_RAW	Valor puro de 0 a 1023
SENSOR_MODE_BOOL	Valor booleano (0 ó 1)
SENSOR_MODE_EDGE	Cuenta números de Transiciones booleanas
SENSOR_MODE_PULSE	Cuenta números de periodos booleanos
SENSOR_MODE_PERCENT	Valor de 0 a 100
SENSOR_MODE_FAHRENHEIT	Grados F – sólo RCX
SENSOR_MODE_CELSIUS	Grados C – sólo RCX
SENSOR_MODE_ROTATION	Rotación (16 tics por revolución) – sólo RCX

Al usar el RCX es normal poner el tipo y el modo al mismo tiempo. La función `SetSensor` hace este proceso un poco más fácil al proporcionar una única función que llamar y establecer un conjunto de combinaciones tipo/modo estándar.

Configuración de sensor	**Tipo**	**Modo**
SENSOR_TOUCH	SENSOR_TYPE_TOUCH	SENSOR_MODE_BOOL
SENSOR_LIGHT	SENSOR_TYPE_LIGHT	SENSOR_MODE_PERCENT
SENSOR_ROTATION	SENSOR_TYPE_ROTATION	SENSOR_MODE_ROTATION
SENSOR_CELSIUS	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_CELSIUS
SENSOR_FAHRENHEIT	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_FAHRENHEIT
SENSOR_PULSE	SENSOR_TYPE_TOUCH	SENSOR_MODE_PULSE
SENSOR_EDGE	SENSOR_TYPE_TOUCH	SENSOR_MODE_EDGE

El RCX proporciona una conversión booleana para todos los sensores –no sólo para los sensores de contacto. Esta conversión booleana se basa normalmente en umbrales preestablecidos para el valor puro. Un valor "bajo" (menos de 460) es un valor booleano de 1. Un valor alto (mayor de 562) es un valor booleano de 0. Esta conversión se puede modificar: al llamar `SetSensorMode` se puede añadir un valor de umbral entre 0 y 31. Si el valor del sensor cambia más que el valor de umbral durante cierto tiempo (3 milisegundos), entonces cambia el estado booleano del sensor. Esto permite que el estado booleano refleje cambios rápidos en el valor puro. Un aumento rápido ocasiona un valor booleano de 0, un descenso rápido es un valor booleano de 1. Incluso cuando un sensor se configura para otro modo (por ejemplo SENSOR_MODE_PERCENT), se lleva a cabo la conversión booleana.

SetSensor (sensor, configuración) **Función - RCX**

Establece el tipo y modo de un sensor dado en una configuración especificada, que debe ser una constante especial conteniendo el tipo y el modo de la información.

```
SetSensor (SENSOR_1, SENSOR_TOUCH);
```

SetSensorType (sensor, tipo) **Función - RCX**

Establece un tipo de sensor, que debe ser una de las de las constantes de tipo de sensor predefinidos.

```
SetSensorType(SENSOR_1, SENSOR_TYPE_TOUCH);
```

SetSensorMode (sensor, modo) **Función - RCX, CM, Spy**

Establece un modo de sensor, que debe ser una de las de las constantes de modo de sensor predefinidos. Se puede añadir, si se desea (sólo al modo RCX) un argumento de umbral para conversión booleana.

```
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW);            // modo puro
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW+10);   // umbral 10
```

ClearSensor(sensor) **Función – Todas**

Borra el valor de un sensor –sólo afecta a los sensores que se configuran para medir una cantidad acumulativa tal como la rotación o un recuento de pulso.

```
ClearSensor(SENSOR_1);
```

3.1.2. Información del sensor

Hay un número de valores que se pueden inspeccionar para cada sensor. Para todos esos valores se debe especificar el sensor por su número de sensor (0, 1 ó 2), y no con la constante correspondiente (ej. `SENSOR_1`).

SensorValue(n) **Valor – Todos**

Devuelve la lectura procesada del sensor para el sensor n, donde n es 0, 1 ó 2. Este es el mismo valor que devuelven los nombres de sensor (ej. `SENSOR_1`).

```
x = SensorValue(0);                       // lee el sensor 1
```

SensorType(n) **Valor – RCX, CM, Scout**

Devuelve el tipo configurado del sensor n, que debe ser 0, 1 ó 2. Sólo tiene tipos configurables de sensor RCX, otros soportes devuelven el tipo el tipo pre-configurado de sensor.

```
x = SensorType(0);
```

SensorMode(n) **Valor – RCX, CyberMaster, Spy**

Devuelve el modo de sensor en uso para el sensor n, que debe ser 0, 1 ó 2.

```
x = SensorMode(0);
```

SensorValueBool(n) **Valor – RCX**

Devuelve el valor booleano del sensor n, que debe ser 0, 1 ó 2. La conversión booleana se hace basándose, o bien en límites preestablecidos, o en un argumento *slope* especificado por medio de `SetSensorMode`.

```
x = SensorValueBool(0);
```

SensorValueRaw(n) **Valor – RCX, Scout, Spy**

Devuelve el valor puro del sensor n, que debe ser 0, 1 ó 2. Los valores puros varían entre 0 y 1023.

```
x = SensorValueRaw(0);
```

3.1.3. Sensor de luz del Scout Scout

En el Scout, `SENSOR_3` se refiere al sensor de luz que viene incorporado. La lectura del valor del sensor de luz (con `SENSOR_3`) devuelve uno de los tres niveles: 0 (oscuro), 1 (normal) o 2 (brillante). Se puede leer el valor puro del sensor con `SensorValueRaw(SENSOR_3),` pero hay que tener en cuenta que una luz más brillante ocasiona un valor puro más bajo. La conversión del valor puro del sensor (entre 0 y 1023) a uno de los tres niveles depende de tres argumentos: *límite superior, límite inferior e histéresis.* El límite inferior es el valor puro más pequeño (más brillante) que aún se considera *normal*. Los valores por debajo del límite más bajo se consideran *brillantes*. El límite superior es el mayor valor puro (más oscuro) que se considera normal. Los valores cerca de este límite se consideran *oscuros*.

Se puede usar la histéresis para impedir que cambie el nivel cuando el valor puro se acerca a uno de los límites. Esto se consigue haciendo que sea un poco más difícil abandonar los estados oscuros y brillantes que entrar en ellos. Específicamente, el límite para moverse de normal a brillante es un poco más bajo que el límite para volver de brillante a normal. La diferencia entre estos dos límites es el margen de histéresis. Lo mismo sucede para la transición entre normal y oscuro.

SetSensorLowerLimit (valor) **Función – Scout**

Establece el valor inferior de la luz del sensor. . El valor puede ser una expresión.

```
SetSensorLowerLimit (100);
```

SetSensorUpperLimit (valor) **Función– Scout**

Establece el valor superior de la luz del sensor. . El valor puede ser una expresión.

```
SetSensorUpperLimit (900);
```

SetSensorHysteresis (valor) **Función – Scout**

Establece la histéresis del sensor de luz. El valor puede ser una expresión.

```
SetSensorHysteresis (20);
```

CalibrateSensor () **Función - Scout**

Lee el valor del sensor de luz, después establece los límites superior e inferior en un 12,5% por encima o por debajo de la lectura actual y establece una histéresis de un 3.12% del valor de la lectura.

```
CalibrateSensor();
```

3.1.4. Sensores del Spybotics Spy

Spybotics utiliza sensores empotrados en lugar de sensors conectados externamente. El sensor de contacto que se encuentra en la parte frontal del ladrillo Spybotics es el SENSOR_1. Está normalmente configurado en modo porcentual, su valor es 0 cuando no está presionado y 100 cuando sí lo está. SENSOR_2 es el sensor de luz (el conector en la parte trasera del ladrillo que se usa para comunicarse con el ordenador). Está normalmente configurado en modo porcentual, valores altos indicarán luz brillante.

3.2. Salidas

3.2.1. Funciones básicas

Todas las funciones que utilizan salidas, establecen como primer argumento un conjunto de salidas. Este valor tiene que ser una constante. Los nombres `OUT_A`, `OUT_B`, y `OUT_C` se usan para identificar las tres salidas. Varias salidas pueden ser combinadas encadenando salidas individuales. Por ejemplo, se usa `OUT_A+OUT_B` para especificar las salidas A y B en una sola instrucción. El valor de las salidas tiene que ser siempre una constante (no puede ser una variable).

Cada salida tiene tres atributos: modo, dirección y potencia. Modo puede ser configurado por medio de `SetOutput(`*`salida,modo`*`)`. El argumento *modo* debe ser una de las siguientes constantes:

Modo	Significado
OUT_OFF	La salida está apagada (el motor no puede girar)
OUT_ON	La salida está encendida (el motor puede funcionar)
OUT_FLOAT	El motor seguirá girando hasta detenerse por si solo

Los otros dos atributos, dirección y potencia, pueden ser configurados en cualquier momento, pero sólo tiene efecto si la salida está encendida. La dirección es configurada mediante el comando `SetDirection(`*salida,dirección*`)`. El argumento *dirección* debe ser una de las siguientes constantes:

Dirección	Significado
OUT_FWD	El motor gira hacia adelante
OUT_REV	El motor gira hacia atrás
OUT_TOGGLE	El motor invierte el sentido de giro

La potencia se puede configurar entre 0 (mínima) y 7 (máxima). Los valores OUT_LOW, OUT_HALF y OUT_FULL están definidos para ser utilizados en la configuración del argumento potencia. La potencia puede ser establecida mediante la

función `SetPower(`*salida,potencia*`)`. Por defecto, los tres motores están configurados a máxima potencia y giro hacia adelante cuando el programa arranca (pero parados).

SetOutput(salidas, modo) **Función - Todos**

Establece la salida en el modo especificado. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`. Modo tiene que ser `OUT_ON`, `OUT_OFF`, o `OUT_FLOAT`.

```
SetOutput(OUT_A + OUT_B, OUT_ON); // Establece A y B encendidos
```

SetDirection(salidas, dirección) **Función - Todos**

Establece la salida en la dirección especificada. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`. Dirección tiene que ser `OUT_FWD`, `OUT_REV`, o `OUT_TOGGLE`.

```
SetDirection(OUT_A, OUT_REV);       // Hace girar A hacia atrás
```

SetPower(salidas, potencia) **Función - Todos**

Establece la potencia del motor especificado. Potencia puede ser una expresión, cuyo resultado debe ser un valor entre 0 y 7. Las constantes `OUT_LOW`, `OUT_HALF`, o `OUT_FULL` también pueden ser usadas.

```
SetPower(OUT_A, OUT_FULL); // A a máxima potencia
SetPower(OUT_B, x);
```

OutputStatus(n) **Valor - Todos**

Devuelve el estado del motor n. Tener en cuenta que n debe ser 0, 1 o 2 – no `OUT_A`, `OUT_B`, o `OUT_C`.

```
x = OutputStatus(0); // Estado de OUT_A
```

3.2.2. Otras funciones

Dado que el control de las salidas es un característica de uso frecuente dentro del programa, se dispone de otras funciones que hacen que trabajar con salidas sea más fácil. Debe tenerse en cuenta que estos comandos no añaden ninguna nueva funcionalidad a la de los comandos `SetOutput` y `SetDirection`. Sólo son interesantes para hacer el programa más conciso.

On(salidas) **Función - Todos**

Establece las salidas especificadas como encendidas. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`.

```
On(OUT_A + OUT_C);                  // Enciende las salidas A y C
```

Off(salidas) **Función - Todos**

Establece las salidas especificadas como apagadas. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`.

```
Off(OUT_A);                         // Apaga la salida A
```

Float(salidas) **Función - Todos**

Establece las salidas especificadas como float. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`.

```
Float(OUT_A);                    // Detiene la salida A sin frenarla
```

Fwd(salidas) **Función - Todos**

Establece el sentido de giro de las salidas especificadas como avance. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`.

```
Fwd(OUT_A);
```

Rev(salidas) **Función - Todos**

Establece el sentido de giro de las salidas especificadas como retroceso. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`.

```
Rev(OUT_A);
```

Toggle(salidas) **Función - Todos**

Invierte el sentido de giro de las salidas especificadas. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`.

```
Toggle(OUT_A);
```

OnFwd(salidas) **Función - Todos**

Establece el sentido de giro de las salidas especificadas como avance y las pone en marcha. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`.

```
OnFwd(OUT_A);
```

OnRev(salidas) **Función - Todos**

Establece el sentido de giro de las salidas especificadas como retroceso y las pone en marcha. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`.

```
OnRev(OUT_A);
```

OnFor(salidas, tiempo) **Función - Todos**

Pone en marcha las salidas especificadas por un determinado tiempo y a continuación las detiene. Salida es uno o más de los valores `OUT_A`, `OUT_B`, y `OUT_C`. Tiempo se mide en incrementos de 10ms (one second = 100) y puede ser una expresión.

```
OnFor(OUT_A, x);
```

3.2.3. Control Global RCX2, Scout

SetGlobalOutput(salidas,modo) **Función - RCX2, Scout, Spy**

Desactiva o vuelve a activar las salidas dependiendo del argumento modo. Si modo es `OUT_OFF`, entonces las salidas se apagarán y se desactivarán. Mientras estén desactivadas cualquier llamada a `SetOutput()` (incluyendo funciones tales como `On()`) serán ignoradas. Si se utiliza el modo `OUT_FLOAT` las salidas estarán establecidas en el modo float antes de desactivarlas. Las salidas pueden volver a ser activadas llamando

`SetGlobalOutput()` y modo `OUT_ON`. Debe tenerse en cuenta que activar una salida no la enciende inmediatamente, sólo permite posteriores llamadas a `SetOutput()`.

```
SetGlobalOutput(OUT_A, OUT_OFF);         //desactiva la salida A
SetGlobalOutput(OUT_A, OUT_ON);          //activa la salida A
```

SetGlobalDirection(salidas, dirección) **Función - RCX2, Scout, Spy**

Invierte o reestablece la dirección de las salidas. El argumento dirección debe ser `OUT_FWD`, `OUT_REV` o `OUT_TOGGLE`. Si `SetGlobalDirection` es `OUT_FWD` el comportamiento de la salida es el normal. Si `SetGlobalDirection` es `OUT_REV` el valor de salida de dirección será el opuesto del que se asigne por las funciones básicas. Si `SetGlobalDirection` es `OUT_TOGGLE` este cambiará entre el comportamiento normal y el comportamiento opuesto.

```
SetGlobalDirection(OUT_A, OUT_REV);            //dirección opuesta
SetGlobalDirection(OUT_A, OUT_FWD);            //dirección normal
```

SetMaxPower(salidas,potencia) **Función - RCX2, Scout, Spy**

Establece el valor máximo de potencia permitido para las salidas. Potencia puede ser una variable, pero debe tener un valor entre `OUT_LOW` y `OUT_FULL`.

```
SetMaxPower(OUT_A, OUT_HALF);
```

GlobalOutputStatus(n) **Función – RCX2, Scout, Spy**

Devuelve la configuración global de la salida del motor `n`. Se debe tener en cuenta que `n` tiene que ser `0`, `1` o `2` –no `OUT_A`, `OUT_B` o `OUT_C`.

```
X = GlobalOutputStatus(0);               //Estado global de OUT_A
```

3.2.4. Salidas de Spybotics

Spybotics tiene dos motores internos. `OUT_A` se refiere al motor derecho y `OUT_B` al izquierdo. `OUT_C` enviará órdenes VLL por medio del LED trasero (el que se utiliza para las comunicaciones con el ordenador). Esto permite a un dispositivo VLL, como el Micro_Scout, ser utilizado como tercer motor del Spubotics. El mismo LED puede ser controlado utilizando las funciones `SendVLL()` y `SetLight()`.

3.3. Sonido

PlaySound(sonido) **Función - Todos**

Ejecuta uno de los 6 sonidos predeterminados del RCX. El argumento `sonido` tiene que ser una constante (excepto en Spybotics, que permite utilizar una variable). Las siguientes constantes están predefinidas para ser usadas con la función `PlaySound()`: `SOUND_CLICK`, `SOUND_DOUBLE_BEEP`, `SOUND_DOWN`, `SOUND_UP`, `SOUND_LOW_BEEP`, `SOUND_FAST_UP`.

```
PlaySound(SOUND_CLICK);
```

PlayTone(frecuencia,duración) **Función - Todos**

Ejecuta un solo tono de la frecuencia y duración especificada. El valor de `frecuencia` es en hercios y puede ser una variable para RCX2, Scout y Spybotics, pero tiene que ser una constante para RCX y CyberMaster. El valor `duración` es en centésimas de segundo y tiene que ser constante.

```
PlayTone(440,50);                  //Ejecuta un La de medio segundo
```

MuteSound() **Función - RCX2, Scout, Spy**

Hace que se dejen ejecutar todos los sonidos y tonos.

```
MuteSound();
```

UnmuteSound() **Función - RCX2, Scout, Spy**

Devuelve al comportamiento normal de sonidos y tonos.

```
UnmuteSound();
```

ClearSound() **Función - RCX2, Spy**

Elimina todos los sonidos pendientes de ser ejecutados en el buffer.

```
ClearSound();
```

SelectSounds(grupo) **Función - Scout**

Selecciona qué grupo de sonidos del sistema deben ser usados. Grupo debe ser una constante.

```
SelectSound();
```

3.4. Display LCD RCX

El RCX tiene 7 modos diferentes de display como se muestra abajo. El predeterminado del RCX es `DYSPLAY_WATCH`.

Modo	Contenido del LCD
DISPLAY_WATCH	Muestra el reloj del sistema
DISPLAY_SENSOR_1	Muestra el valor del sensor 1
DISPLAY_SENSOR_2	Muestra el valor del sensor 2
DISPLAY_SENSOR_3	Muestra el valor del sensor 3
DISPLAY_OUT_A	Muestra la configuración de la salida A
DISPLAY_OUT_B	Muestra la configuración de la salida B
DISPLAY_OUT_C	Muestra la configuración de la salida C

El RCX2 añade un octavo modo: `DISPLAY_USER`. Este modo lee continuamente un valor fuente y lo muestra en pantalla. Puede mostrar un punto decimal en cualquier posición entre las cifras. Esto permite emular el trabajo con fracciones aunque todos los valores estén almacenados como enteros. Por ejemplo, la siguiente función hará mostrar el valor 1234, mostrando dos cifras después del punto decimal, haciendo que aparezca "12.34" en el LCD.

```
SetUserDisplay(1234,2);
```

El siguiente programa ilustra la actualización del display:

```
task main()
{
    ClearTimer(0);
    SetUserDisplay(Timer(0),0);
    Until(false);
}
```

Dado que el modo `SetUserDysplay` actualiza constantemente el LCD, hay algunas restricciones en el código fuente. Si se usa una variable debe ser asignada a una variable global. La mejor manera para asegurarse de que es así es declararla como variable global. También pueden producirse otros efectos extraños. Por ejemplo, si se está mostrando una variable y se ejecuta un cálculo en el que el resultado es la variable, es posible que el display muestre algunos resultados intermedios:

```
int x;
task main()
{
    SetUserDisplay(x,0);
    while(true)
    {
            // El display puede mostrar durante un instante 1
            x = 1 + Timer(0);
    }
}
```

SelectDisplay(modo) **Función - RCX**

Selecciona un modo de display.

```
SelectDisplay(DISPLAY_SENSOR_1);         // muestra el sensor 1
```

SetUserDisplay(valor,precisión) **Función - RCX2**

Establece que el display LCD muestre continuamente un valor especificado. Precisión especifica el numero de dígitos a la derecha del punto decimal. Una precisión de `0` no muestra punto decimal.

```
SetUserDisplay(Timer(0),0);              // muestra el temporizador 0
```

3.5. Comunicaciones

3.5.1. Mensajes RCX, Scout

El RCX y el Scout pueden enviar y recibir mensajes simples utilizando los infrarrojos. Un mensaje puede tener un valor desde 0 hasta 255, pero no se recomienda utilizar el mensaje 0. El último mensaje recibido es guardado y puede accederse a él mediante `Message()`. Si no ha sido recibido ningún mensaje, `Message()` devolverá el valor 0. Debe tenerse en cuenta, que debido a la naturaleza de la comunicación mediante infrarrojos, no se podrán recibir mensajes mientras un mensaje se esté transmitiendo.

ClearMessage(valor,precisión) **Función - RCX, Scout**

Borra el buffer de mensajes. Esto facilita la detección de un mensaje recibido ya que entonces el programa puede esperar a que `Message()` no sea cero:

```
ClearMessage();              // Borra los mensajes recibidos
until(Message() > 0);        //Esperar al siguiente mensaje
```

SendMessage(mensaje) **Función - RCX, Scout**

Envía un mensaje por infrarrojos. `Mensaje` puede ser una expresión, pero el RCX sólo puede enviar mensajes con un valor entre 0 y 255, por lo tanto sólo los 8 bits menores del argumento serán usados.

```
SendMessage(3);              // enviar mensaje 3
SendMessage(259);      // otra manera de enviar el mensaje3
```

SetTxPower(potencia) **Función – RCX, Scout**

Establece la potencia para la transmisión por infrarrojos. `Potencia` tiene que ser una de estas constantes: `TX_POWER_LO` o `TX_POWER_HI`

3.5.2. Serie RCX2, Scout

El RCX2 puede transmitir datos serie por el puerto de infrarrojos. Antes de enviar datos, la configuración de la comunicación y de los paquetes tiene que ser especificada. Entonces, para cada transmisión, los datos deben ser colocados en el buffer de transmisión y entonces usar la función `SendSerial()` para enviarlos. La configuración de comunicación es establecida mediante `SetSerialComm()` y determina cómo son enviados los bits mediante el puerto de infrarrojos. Los valores posibles se muestran a continuación:

Opción	Efecto
SERIAL_COMM_DEFAULT	Configuración predeterminada
SERIAL_COMM_4800	4800 baudios
SERIAL_COMM_DUTY25	25% duty cycle
SERIAL_COMM_76KHZ	76 kHz carrier

Por defecto, está configurado a enviar datos a 2400 baudios usando un duty cycle del 50% en un carrier de 38 kHz. Para especificar opciones múltiples (como a 4800 baudios con un duty cycle del 25%), se combinan las opciones individuales utilizando OR bit a bit (`SERIAL_COMM_4800 | SERIAL_COMM_DUTY25`). La configuración de los paquetes se establece con `SetSerialPacket` y controla la manera como se ensamblan los bytes en paquetes. Los valores posible se muestran abajo.

Opción	Efecto
SERIAL_PACKET_DEFAULT	Sin formato de paquete, sólo los bits de datos
SERIAL_PACKET_PREAMBLE	Envía un preámbulo del paquete
SERIAL_PACKET_NEGATED	Envía cada byte con su complementario
SERIAL_PACKET_CHECKSUM	Incluye un checksum para cada paquete
SERIAL_PACKET_RCX	Formato estándar del RCX (preámbulo, datos negados y checksum)

Se debe tener en cuenta que los paquetes negados siempre incluyen un checksum, por lo tanto la opción `SERIAL_PACKET_CHECKSUM` sólo tiene significado cuando `SERIAL_PACKET_NEGATED` no ha sido especificada. Igualmente el preámbulo, negados y checksum están implícitos en `SERIAL_PACKET_RCX`. El buffer de transmisión puede guardar hasta 16 bytes de datos. Por ejemplo, el siguiente código envía dos bytes (0x12 y 0x34) al puerto serie:

```
SetSerialComm(SERIAL_COMM_DEFAULT);
SetSerialPacket(SERIAL_PACKET_DEFAULT);
SetSerialData(0,0x12);
SetSerialData(1,0x34);
SendSerial(0,2);
```

SetSerialComm(configuración) **Función - RCX2**

Establece la configuración de la comunicación que determina de que modo son enviados los bits por el infrarrojos.

```
SetSerialComm(SERIAL_COMM_DEFAULT);
```

SetSerialPacket(configuración) **Función - RCX2**

Establece la configuración de los paquetes que determina de que modo los bytes son ensamblados en paquetes.

```
SetSerialPacket(SERIAL_PACKET_DEFAULT);
```

SetSerialData(n,valor) **Función RCX2**

Coloca un byte de datos en el buffer de transmisión. `n` es el índice del byte a establecer (0-15), y `valor` puede ser una expresión.

```
SetSerialData(3,x); // establece el byte 3 en x
```

SerialData(n) **Función - RCX2**

Devuelve el valor del byte del buffer de transmisión (no los datos recibidos). `n` tiene que ser una constante entre 0 y 15.

```
X = SerialData(7);// lee el byte #7
```

SendSerial(comienzo, contador) **Función - RCX2**

Utiliza el contenido del buffer de transmisión para construir un paquete y enviarlo por infrarrojos (de acuerdo con la configuración actual de paquetes y comunicación). `Comienzo` y `contador` son constantes que especifican el primer byte y el número de bytes dentro del buffer que se deben enviar

```
SendSerial(0,2);// enviar los dos primeros bytes del buffer
```

3.5.3. VLL (Enlace de luz visible) Scout

SendVLL(valor) **Función – Scout, Spy**

Envía un comando VLL que puede ser usado para comunicarse con el MicroScout o el CodePilot. Los comandos específicos VLL están descritos en el SDK del Scout.

```
SendVLL(4); // Envía el comando VLL #4
```

3.6. Temporizadores

Los diferentes destinos ofrecen temporizadores independientes con una resolución de 100 ms (10 tics por segundo). El Scout dispone de 3 temporizadores, mientras el RCX, el Cybermaster y el Spybotics de 4. Los temporizadores van desde el tic 0 hasta el tic 32767 (alrededor de 55 minutos). El valor del temporizador puede ser leído usando `Timer(n)`, donde `n` es una constante que determina qué temporizador usar (0-2 para el Scout, 0-3 para el resto). El RCX2 y el Spybotics poseen la característica de leer el mismo temporizador con mayor resolución usando `FastTimer(n)`, que devuelve el valor del temporizador con una resolución de 10 ms (100 tics por segundo).

ClearTimer(n) **Función - Todos**

Pone a cero el temporizador especificado.

```
ClearTimer(0);
```

Timer(n) **Función - Todos**

Devuelve el valor actual del temporizador especificado (con una resolución de 100 ms).

```
x = Timer(0);
```

SetTimer(n,value) **Función – RCX2, Spy**

Configura el temporizador con un valor especificado (que puede ser una expresión).

```
SetTimer(0,x);
```

FastTimer(n) **Función - RCX2, Spy**

Devuelve el valor actual del temporizador especificado con una resolución de 10 ms.

```
x = FastTimer(0);
```

3.7. Contadores RCX2, Scout, Spy

Los contadores son como variables sencillas que pueden ser incrementadas decrementadas y borradas. El Scout dispone de dos contadores (0 y 1), mientras que el RCX2 y Spybotics disponen de tres (0, 1, 2). En el caso del RCX2, estos contadores se solapan con direcciones de almacenaje globa 0-2, por lo tanto, si van a ser usadas como contadores habrán de ser reservadas con #pragma para evitar que NQC las utilice como una variable común. Por ejemplo, si se desea utilizar el contador 1:

```
#pragma reserve 1
```

ClearCounter(n) **Función - RCX2, Scout, Spy**

Pone a cero el contador n. n tiene que ser 0 ó 1 para el Scout, 0-2 para el RCX2 y Spybotics.

```
Clear Counter(1);
```

IncCounter(n) **Función - RCX2, Scout, Spy**

Incrementa el contador n en 1. n tiene que ser 0 ó 1 para el Scout, 0-2 para el RCX2 y Spybotics.

```
IncCounter(1);
```

DecCounter(n) **Función - RCX2, Scout, Spy**

Decrementa el contador n en 1. n tiene que ser 0 ó 1 para el Scout, 0-2 para el RCX2 y Spybotics.

```
DecCounter(1);
```

Counter(n) **Función - RCX2, Scout, Spy**

Devuelve el valor del contador n. n tiene que ser 0 ó 1 para el Scout, 0-2 para el RCX2 y Spybotics.

```
x = Counter(1);
```

3.8. Control de Acceso RCX2, Scout, Spy

El control de acceso es implementado principalmente por medio de las declaraciones `acquire`. La función `SetPriority` puede ser usada para establecer la prioridad de una función, y las siguientes constantes pueden ser usadas para especificar los recursos en una declaración `acquire`. Debe tenerse en cuenta que la definición de los recursos sólo está disponible en el RCX2.

Constante	Recurso
ACQUIRE_OUT_A	Salidas
ACQUIRE_OUT_B	
ACQUIRE_OUT_C	
ACQUIRE_SOUND	Sonido
ACQUIRE_LED	LEDs (sólo Spybotics)
ACQUIRE_USER_1	Definidas por el usuario – sólo en RCX2
ACQUIRE_USER_2	
ACQUIRE_USER_3	
ACQUIRE_USER_4	

SetPriority(p) **Función - RCX2, Scout, Spy**

Establece la prioridad de una función a p, que debe ser constante. RCX2 soporta prioridades 0-255, mientras el Scout soporta prioridades 0-7. Debe tenerse en cuenta que a números menores, prioridad mayor.

```
SetPriority(1);
```

3.9. Eventos RCX2, Scout

Aunque RCX2, Scout y Spybotics comparten un mecanismo común de eventos, el RCX2 y Spybotics disponen de 16 eventos completamente configurables, mientras que el Scout dispone de 15 eventos predefinidos. Las únicas funciones comunes estos destinos son los comandos para inspeccionar y forzar eventos.

ActiveEvents(tarea) **Valor - RCX2, Scout, Spy**

Devuelve los eventos que se han producido en una tarea dada.

```
x = ActiveEvents(0);
```

CurrentEvents() **Valor - RCX2, Scout, Spy**

Devuelve los eventos que se han producido en la tarea actual.

```
x = CurrentEvents();
```

Event(eventos) **Valor - RCX2, Scout, Spy**

Activa manualmente un evento. Esto puede ser útil para probar el tratamiento de eventos de un programa o, en otros casos, para simular un evento basado en otros argumentos. Debe tenerse en cuenta que la especificación de eventos difiere un poco entre el RCX2 y el Scout. RCX2 usa la macro EVENT_MASK para computar una máscara de evento, mientras que el Scout las tiene predefinidas.

```
Event(EVENT_MASK(3)); // Activa un evento en el RCX2
Event(EVENT_1_PRESSED)); // Activa un evento en el Scout
```

3.9.1. Eventos del RCX2 RCX2, Spy

Nota: Spybotics events appear to be very similar to RCX2 events, although very little testing has been done for the NQC API y Spybotics. La información siguiente ha sido escrita desde la perspectiva del RCX2, y no ha sido todavía actualizada para Spybotics.

El RCX2 y Spybotics ofrecen un sistema de eventos extremadamente flexible. Hay 16 eventos, cada uno de ellos se relaciona con una de las diferentes fuentes de eventos (el estímulo que puede hacer disparar el evento) y el tipo de evento (el criterio para que se dispare).

Otros argumentos pueden ser especificados dependiendo del tipo de evento. Para todas las llamadas desde una función, un evento se identifica por su número de evento –una constante entre 0 y 15. Fuentes de eventos son los sensores, temporizadores, contadores o el buffer de mensajes. Un evento es configurado llamando a `SetEvent(`***`evento,fuente,tipo`***`)`, donde evento es un número constante (0-15), fuente es la fuente del evento, y tipo es uno de los tipos que se muestran a continuación (algunas combinaciones de fuentes y tipos no son posibles).

Tipo de Evento	**Condición**	**Fuente del Evento**
EVENT_TYPE_PRESSED	El valor cambia a *on*	Sólo sensores
EVENT_TYPE_RELESASED	El valor cambia a *off*	Sólo sensores
EVENT_TYPE_PULSE	El valor cambia de *off* a *on* y otra vez a *off*	Sólo sensores
EVENT_TYPE_EDGE	El valor cambia de *on* a *off* o viceversa	Sólo sensores
EVENT_TYPE_FASTCHANGE	El valor varía rápidamente	Sólo sensores
EVENT_TYPE_LOW	El valor cambia a *low*	Cualquiera
EVENT_TYPE_NORMAL	El valor cambia a *normal*	Cualquiera
EVENT_TYPE_HIGH	El valor cambia a *high*	Cualquiera
EVENT_TYPE_CLICK	El valor cambia de *low* a *high* y otra vez a *low*	Cualquiera
EVENT_TYPE_DOUBLECLICK	Dos clics durante un determinado tiempo	Cualquiera
EVENT_TYPE_MESSAGE	Nuevo mensaje recibido	Sólo Message()

Los primeros cuatro eventos se basan en el valor booleano de un sensor, así que son los más útiles con sensores de contacto. Por ejemplo, para configurar el evento #2 que se dispare cuando el sensor de contacto del puerto 1 es presionado, puede ser así:

```
SetEvent(2,SENSOR_1,EVENT_TYPE_PRESSED);
```

Cuando se quiera usar `EVENT_TYPE_PULSE` o `EVENT_TYPE_EDGE`, el sensor tiene que estar configurado como `SENSOR_MODE_PULSE` o `SENSOR_MODE_EDGE` respectivamente.

`EVENT_TYPE_FASTCHANGE` debe ser usado con sensores que hayan sido configurados con un argumento *slope*. Cuando el valor *raw* cambie más rápido que el argumento *slope* un evento `EVENT_TYPE_FASTCHANGE` se disparará.

Los siguientes tres tipos (`EVENT_TYPE_LOW`, `EVENT_TYPE_NORMAL` y `EVENT_TYPE_HIGH`) convierten el valor fuente del evento en una de las tres gamas (*low [baja]*, *normal* o *high[alta]*), y el evento se disparará cuando el valor pase de una gama a la otra. Las gamas son definidas por *lower limit* (límite inferior) y *upper limit* (límite superior) para el evento. Cuando el valor fuente es menor que el límite inferior, la fuente será considerada *baja*. Cuando el valor fuente sea mayor que el límite superior, la fuente será considerada *alta*. La fuente será *normal*, cuando esté entre los dos límites.

El siguiente evento configura el evento #3 para que se dispare cuando el valor del sensor en el puerto 2 esté en la gama *alta*. El límite superior está establecido en 80 y el límite inferior en 50. Está configuración es un ejemplo de como un evento puede dispararse cuando el sensor de luz detecta luz clara.

```
SetEvent(3,SENSOR_2,EVENT_TYPE_HIGH);
SetLowerLimit(3,50);
SetUpperLimit(3,80);
```

El argumento *hysteresis* puede ser usado para hacer que las transiciones sean más estables en los casos en que la el valor varia. Histéresis funciona haciendo que la transición de *baja* a *normal* sea un poco más amplia que la transición de *normal* a *baja*. Así hace que sea más fácil entrar en la gama *baja* que salir de ella. Lo mismo se aplica a la transición de *normal* a *alta*.

Una transición de *baja* a *alta* y otra vez a *baja* hará disparar el evento `EVENT_TYPE_CLICK` siempre que la secuencia completa sea más rápida que el tiempo de clic del evento. Si se producen dos clics seguidos y el tiempo entre ellos es inferior al

tiempo de clic, se disparará un evento `EVENT_TYPE_DOUBLECLICK`. El sistema también se mantiene al tanto del número total de clics para cada evento.

El ultimo tipo de evento, `EVENT_TYPE_MESSAGE`, sólo es valido cuando `Message()` es usado como la fuente de un evento. Este evento se disparará cuando un nuevo mensaje llegue (incluso si su valor es el mismo que el del mensaje anterior).

Monitorizar instrucciones y algunas funciones API (como `ActiveEvents()` o `Event()`) requiere manipular múltiples eventos. Esto se hace convirtiendo cada número de evento en una máscara de evento, y entonces combinando las máscaras con una OR bit a bit. La macro `EVENT_MASK(evento)` convierte un número de evento en una máscara. Por ejemplo, para mostrar los eventos 2 y 3 puede utilizarse la siguiente instrucción:

```
monitor(EVENT_MASK(2) | EVENT_MASK(3))
```

SetEvent(evento, fuente, tipo) — Función – RCX2, Spy

Configura un evento (un número entre 0 y 15) para usar la fuente y el tipo especificado. Ambos eventos y tipos tienen que ser constantes, y fuente debe ser la expresión de una fuente que se quiere utilizar.

```
SetEvent(2, Timer(0), EVENT_TYPE_HIGH);
```

ClearEvent(evento) — Valor – RCX2, Spy

Borra la configuración del evento especificado. Esto permite que no se dispare hasta que sea otra vez configurado.

```
ClearEvent(2); //borrar evento #2
```

ClearAllEvents() — Valor – RCX2, Spy

Borra la configuración de todos los eventos.

```
ClearAllEvents();
```

EventState(evento) — Valor – RCX2, Spy

Devuelve el estado del evento dado. Los estados son 0: Low (bajo), 1: Normal, 2: High (alto), 3: Undefined (indefinido), 4: Start calibrating (inicio de calibración), 5: Calibrating process (en proceso de calibración).

```
X = EventState(2);

ClearAllEvents();
```

CalibrateEvent(evento, inferior, superior, histéresis) — Función – RCX2, Spy

Calibra el evento tomando la lectura actual del sensor y aplicándole los ratios especificados de *inferior*, *superior* e *histéresis* para determinar los límites y el valor de la histéresis. Las fórmulas para la calibración dependen del tipo de sensor y están explicadas en el LEGO SDK. La calibración no es instantánea. EventState() puede ser comprobado para determinar cuándo se ha completado la calibración (normalmente 50ms).

```
CalibrateEvent(2,50,50,20);

Until(EventState(2) != 5);              //esperar a la calibración
```

SetUpperLimit(evento, limite) **Valor – RCX2, Spy**

Establece el límite superior para el evento, donde evento es un número de evento constante y límite puede ser cualquier expresión.

```
SetUpperLimit(2,x);//establece el límite superior para #2 en x
```

UpperLimit() **Valor – RCX2, Spy**

Devuelve el valor del límite superior del número de evento especificado.

```
x = UpperLimit(2);// Obtener el límite superior del evento #2
```

SetLowerLimit(evento, límite) **Valor – RCX2, Spy**

Establece el límite inferior para el evento, donde evento es un número de evento constante y límite puede ser cualquier expresión.

```
SetLowerLimit(2,x);//establece el límite inferior para #2 en x
```

LowerLimit() **Valor – RCX2, Spy**

Devuelve el valor del límite inferior del número de evento especificado.

```
x = LowerLimit(2);// Obtener el límite inferior del evento #2
```

SetHysteresis(evento, valor) **Valor – RCX2, Spy**

Establece la histéresis del evento, donde evento es un número de evento constante y valor puede ser cualquier expresión.

```
SetHysteresis(2,x);
```

Hysteresis(evento) **Valor – RCX2, Spy**

Devuelve el valor de la histéresis del número de evento especificado.

```
x = Hysteresis(2);
```

SetClickTime(evento,valor) **Función – RCX2, Spy**

Establece el tiempo de clic para el evento, donde evento es un número de evento constante y valor puede ser cualquier expresión. El tiempo se especifica en incrementos de 10 ms, entonces un segundo tendrá el valor de 100.

```
SetClickTime(2,x);
```

ClickTime(evento) **Valor – RCX2, Spy**

Devuelve el valor del tiempo de clic para el número de evento especificado.

```
x = ClickTime(2);
```

SetClickCounter(evento,valor) **Función – RCX2, Spy**

Establece el contador de clic para el evento, donde evento es una número de evento constante y valor puede ser cualquier expresión.

```
SetClickCounter(2,x);
```

ClickCounter(evento) **Valor – RCX2, Spy**

Devuelve el valor del contador de clic para el número de evento especificado.

```
x = ClickCounter(2);
```

3.9.2. Eventos del Scout Scout

El Scout ofrece 15 eventos, cada uno de los cuales tiene un significado predefinido como se muestra en la tabla inferior.

Nombre del Evento	**Condición**
EVENT_1_PRESSED	Sensor 1 pasa a estar presionado
EVENT_1_RELEASED	Sensor 1 pasa a no estar presionado
EVENT_2_PRESSED	Sensor 2 pasa a estar presionado
EVENT_2_RELEASED	Sensor 2 pasa a no estar presionado
EVENT_LIGHT_HIGH	Sensor de luz *alto*
EVENT_LIGHT_NORMAL	Sensor de luz *normal*
EVENT_LIGHT_LOW	Sensor de luz *bajo*
EVENT_LIGHT_CLICK	De *bajo* a *alto* y otra vez *a bajo*
EVENT_LIGHT_DOUBLECLICK	Dos clics
EVENT_COUNTER_0	Contador 0 por encima del límite
EVENT_COUNTER_1	Contador 1 por encima del límite
EVENT_TIMER_0	Timer 0 por encima del límite
EVENT_TIMER_1	Timer 1 por encima del límite
EVENT_TIMER_2	Timer 2 por encima del límite
EVENT_MESSAGE	Nuevo mensaje recibido

Los primeros cuatro eventos se disparan mediante sensores de contacto conectados a los dos puertos de sensores. `EVENT_LIGHT_HIGH`, `EVENT_LIGHT_NORMAL` y `EVENT_LIGHT_LOW` se disparan cuando el valor del sensor de luz cambia de una gama a otra. Las gamas son definidas mediante `SetSensorUpperLimit`, `SetSensorLowerLimit`, y `SetSensorHysteresis` que han sido descritos anteriormente.

`EVENT_LIGHT_CLICK` y `EVENT_LIGHT_DOUBLECLICK` también se disparan mediante el sensor de luz. Un clic es la transición de *bajo* a *alto* y de vuelta a *bajo* dentro de un determinado periodo de tiempo llamado *tiempo de clic*.

Cada contador tiene un límite de contador. Cuando el contador supera este límite, `EVENT_COUNTER_0` o `EVENT_COUNTER_1` se disparan. Los temporizadores también tiene un límite, y generan `EVENT_TIMER_0`, `EVENT_TIMER_1`, y `EVENT_TIMER_2`.

`EVENT_MESSAGE` se dispara cuando un nuevo mensaje es recibido por el infrarrojos.

SetSensorClickTime(valor) **Función – Scout**

Establece el *tiempo de clic* utilizado para generar eventos del sensor de luz. El valor debe ser especificado en incrementos de 10 ms, y puede ser una expresión.

```
SetSensorClickTime(x);
```

SetCounterLimit(n,valor) **Función – Scout**

Establece el limite para el contador n. n debe ser 0 ó 1 y valor puede ser una expresión.

```
SetCounterLimit(0,100); //Establece el limite del contador 0 en 100
```

SetTimerLimit(n,valor) **Función – Scout**

Establece el limite para el *temporizador* n. N debe ser 0, 1 o 2, y valor puede ser una expresión.

```
SetTimerLimit(1,100); //Establece el temporizador 0 en 100
```

3.10. Registro de datos RCX

El RCX contiene un registro de datos que puede ser usado para almacenar valores de los sensores, temporizadores, variables y del reloj del sistema. Antes de añadir datos, el registro de datos necesita ser creado por medio del comando `CreateDatalog(tamaño)`. El argumento `tamaño` debe ser una constante y determina cuantos datos pueden ser almacenados:

```
CreateDatalog(100);          // registro de datos de 100 valores
```

Pueden añadirse valores al registro de datos usando `AddToDatalog(valor)`. Cuando el registro de datos se transmite al ordenador este mostrará ambos, el valor y la fuente del valor (temporizador, variable, etc). El registro de datos soporta directamente las siguientes fuentes de datos: temporizadores, valores de sensores, variables, y el reloj de sistema. Otros tipos de datos (como constantes o números aleatorios) también pueden ser almacenados, pero en este caso NQC primero asignará el valor a una variable y luego la almacenará. Los valores podrán ser fielmente almacenados en el datalog, pero el origen del dato quedará desvirtuado.

```
AddToDatalog(Timer(0));      // Añadir temporizador 0 al registro de
                             // datos
AddToDatalog(x);       // Añadir variable x
AddToDatalog(7);       // Añadir 7 – se verá como una variable
```

El RCX no puede leer por sí solo los valores del registro de datos. El registro de datos debe ser transferido a un ordenador. Las especificidades de la transferencia del registro de datos depende del medio en el cual se esté utilizando NQC. Por ejemplo, en la línea de comandos de NQC los siguientes comandos transferirán y imprimirán el registro de datos:

```
nqc –datalog
nqc –datalog_full
```

CreateDatalog(tamaño) **Función – RCX**

Crea un registro de datos del tamaño especificado (que debe ser constante). Un tamaño de 0 elimina el anterior sin crear uno nuevo.

```
CreateDatalog(100); //registro de datos de 100 puntos
```

AddToDatalog(valor) **Función – RCX**

Añade el valor, que puede ser una expresión, al registro de datos. Si el registro de datos está lleno la llamada no tiene efecto.

```
AddToDatalog(x);
```

UploadDatalog(comienzo, contador) **Función – RCX**

Inicia y transmite un número de datos igual a contador, comenzando por el valor de *comienzo*. Este comando no es muy útil ya que el ordenador es el que normalmente comienza la transmisión.

```
UploadDatalog(0,100);   //transmite el registro de los primeros
                        // 100 puntos
```

3.11. Características generales

Wait(tiempo) **Función – Todos**

Hace parar una tarea durante un tiempo especificado (en centésimas de segundo). El argumento tiempo puede ser una expresión o una constante:

```
Wait(100);              //Espera durante un segundo
Wait(Random(100));            //Espera durante un tiempo aleatorio de
                              //hasta 1 segundo.
```

StopAllTasks() **Función – Todos**

Detiene todas las tareas que están ejecutándose. Esto hará parar el programa completamente, por lo tanto, cualquier comando que siga a éste será ignorado.

```
StopAllTasks();         //detiene el programa
```

Random(n) **Valor – Todos**

Devuelve un número aleatorio entre 0 y n. n debe ser una constante.

```
x = Random(10);
```

SetRandomSeed(n) **Función – RCX2, Spy**

Preselecciona el generador de números aleatorios con n. n puede ser una expresión.

```
SetRandomSeed(x);             //preselecciona con el valor de x
```

SetSleepTime(minutos) **Función – Todos**

Establece el numero de minutos (que debe ser constante) que estará activo hasta que se duerma. Especificando 0 minutos deshabilita esta opción.

```
SetSleepTime(5); //Dormirse al cabo de 5 minutos
SetSleepTime(0); //Deshabilita el tiempo para que se duerma
```

SleepNow(n) **Función – Todos**

Fuerza a que se duerma. Sólo funciona si el tiempo para que se duerma no es cero.

```
SleepNow(); //Ir a dormir
```

3.12. Características específicas del RCX

Program() **Valor – RCX**

Número del programa que está seleccionado.

```
x = Program();
```

SelectProgram(n) **Función – RCX2**

Selecciona el programa especificado para que comience a ejecutarse. Hay que tener en cuenta que los programas están numerados del 0 al 4 (no de 1 a 5 como se muestra en el LCD).

```
SelectProgram(3);
```

BatteryLevel() **Valor – RCX2**

Devuelve el nivel de batería en milivolts.

```
x = BatteryLevel();
```

FirmwareVersion() **Valor – RCX2**

Devuelve la versión del firmware como un entero. Por ejemplo, la versión 3.2.6 es 326.

```
x = FirmwareVersion();
```

Watch() **Valor – RCX**

Devuelve el valor del reloj del sistema en minutos.

```
x = Watch();
```

SetWatch(horas,minutos) **Función – RCX**

Establece en el reloj de sistema un número específico de horas y minutos. Horas debe ser un número constante entre 0 y 23 ambos incluidos. Minutos debe ser una constante entre 0 y 59 ambos incluidos.

```
SetWatch(3,15); //Establece el reloj en 3:15
```

3.13. Características específicas del Scout

SetScoutRules(movimiento, tacto, luz, tiempo, fx) **Función – Scout**

Establece las reglas usadas por el Scout en el modo *stand-alone*.

ScoutRules(n) **Valor – Scout**

Devuelve el valor de la configuración de una de las reglas. n debe ser un número constante entre 0 y 4.

```
x = ScoutRules(1);// Obtener la regla #1
```

SetScoutMode(modo) **Función – RCX**

Pone el Scout en modo *stand-alone*(0) o *power* (1). Dentro de un programa sólo tiene sentido utilizarla para ponerlo en modo *stand-alone*, ya que para que pueda ejecutarse un programa NQC debe estar en modo *power*.

SetEventFeedback(eventos) **Función – Scout**

Establece qué eventos deben ser acompañados por un sonido.

```
SetEventFeedback(EVENT_1_PRESSED);
```

EventFeedback() **Valor – Scout**

Devuelve los eventos que están acompañados de un sonido.

```
x = eventFeedback();
```

SetLight(modo) **Función – Scout**

Controla el LED del Scout. Modo debe ser `LIGHT_ON` o `LIGHT_OFF`.

```
SetLight(LIGHT_ON);// Encender el LED
```

3.14. Características específicas del CyberMaster

CyberMaster ofrece nombres alternativos para los sensores: SENSOR_L, SENSOR_M y SENSOR_R. También ofrece nombres alternativos para las salidas: OUT_L, OUT_R, OUT_X. Además, los dos motores internos tienen tacómetros, que miden clics y velocidad conforme el motor gira. Hay unos 50 clics por revolución. Los tacómetros pueden ser usados, por ejemplo, para crear un robot que pueda detectar si ha chocado con un objeto sin usar un sensor externo. Los tacómetros tienen un valor máximo de 32767 y no diferencian entre ambos sentidos. También contará si se gira el motor con la mano, incluso si no hay ningún programa funcionando.

Drive(motor0,motor1) **Función – CyberMaster**

Enciende ambos motores en el nivel de potencia especificado. Si la potencia es negativa, el motor girará en sentido inverso. Equivale al siguiente código:

```
SetPower(OUT_L,abs(power0));
SetPower(OUT_R,abs(power1));
if(power0<0)
   {SetDirection(OUT_L,OUT_REV)}
else
   {SetDirection(OUT_L,OUT_FWD)}
if(power1<0)
   {SetDirection(OUT_R,OUT_REV)}
else
```

```
    {SetDirection(OUT_R,OUT_FWD)}
SetOutput(OUT_L+OUT_R,OUT_ON);
```

OnWait(motor,n,tiempo) **Función – CyberMaster**

Enciende los motores especificados, todos a la misma potencia y entonces espera durante el tiempo dado. El tiempo se da en décimas de segundo, con un máximo de 255 (25.5 segundos). Equivale al siguiente código:

```
SetPower(motores,abs(power));
if(power<0)
    {SetDirection(motores,OUT_REV)}
else
    {SetDirection(motores,OUT_FWD)}
SetOutput(motores,OUT_ON);
Wait(tiempo*10);
```

OnWaitDifferent(motores,n0,n1,n2,tiempo) **Función – CyberMaster**

Como OnWait(), excepto que se pueden dar diferentes potencias para cada motor.

ClearTachoCounter(motores) **Función – CyberMaster**

Pone a cero el tacómetro del motor especificado.

TachoCount(n) **Valor – CyberMaster**

Devuelve el valor del tacómetro del motor especificado.

TachoSpeed(n) **Valor – CyberMaster**

Devuelve el valor de la velocidad del tacómetro del motor especificado. La velocidad es casi constante para un motor en vacío a cualquier velocidad, con un valor máximo de 90 (este será menor conforme las baterías pierden potencia). El valor disminuye conforme la carga en el motor aumenta. Un valor de 0 indica que el motor está bloqueado.

ExternalMotorRunning() **Valor – CyberMaster**

Esto es en realidad la medida del estado actual del motor Los valores devueltos tienden a fluctuar un poco, pero, en general, son los siguientes para un motor sin carga:

0	el motor está en "*floating*"
1	el motor está apagado
<=7	el motor gira alrededor de ese nivel de potencia. Aquí es donde el valor fluctúa más. En cualquier caso, debe saber qué nivel de potencia ha establecido en el motor. El valor incrementa conforme la carga en el motor aumenta, y un valor entre 260 y 300 indica que el motor está bloqueado.

AGC() **Valor – CyberMaster**

Devuelve el valor del AGC (control automático de ganancia) en el receptor de radiofrecuencia. Esto puede ser usado para dar una medida de la distancia un poco inexacta entre el CyberMaster y el transmisor de radiofrecuencia.

```
x = AGC();
```

4. Detalles técnicos

Esta sección explica algunas de las características de bajo nivel de NQC. En general, estos mecanismos deben ser usados solamente como último recurso, ya que pueden cambiar en las futuras versiones. La mayoría de los programadores nunca necesitarán utilizar estas características descritas a continuación –son utilizadas principalmente en la creación del archivo API de NQC.

4.1. La instrucción asm

La instrucción asm es utilizada para definir la mayoría de las llamadas a la API de NQC. La sintaxis de esta instrucción es:

```
asm {ítem1, ítem2, ... ítemN}
```

donde ítem es uno de los siguientes

```
expresión_constante
&expresión
&expresión: restricción
```

La instrucción emite simplemente el valor de cada uno de los ítem como bytecodes "*raw*" (los 8 bits menores del valor constante). Por ejemplo, el archivo API define la siguiente función:

```
Void ClearMessage() {asm{0x90};}
```

Cuando ClearMessage() es llamado por un programa, el valor 0x90 se emite como un *bytecode*.

Muchas funciones API toman argumentos y estos argumentos deben ser codificados en una dirección apropiada para el intérprete de *bytecode*. En el caso más general, una dirección contiene el código fuente seguido de un valor de dos *bytes* (de menor importancia el primer *byte*): Los códigos fuentes están explicados en la documentación del SDK disponible de LEGO.

Sin embargo, es deseable codificar el valor de otro modo –por ejemplo utilizar sólo un valor de un solo byte después del código fuente, omitir el código fuente, o permitir usar sólo ciertas fuentes. Una restricción puede ser usada para controlar cómo está formada la dirección. Una restricción es un valor constante de 32 bit. Los 24 bits menores forman una máscara indicando qué fuentes son válidas (el bit 0 debe ser establecido para permitir la fuente 0, etc).

Los 8 bits superiores incluyen el formato de los *flags* para la dirección. Debe tenerse en cuenta que cuando no se especifica una restricción, es lo mismo que usar el restrictor 0 (ninguna restricción en la fuente, y el formato de la fuente seguido de un valor de dos bytes). El archivo API define las siguientes constantes que pueden ser usadas para usar restrictores:

```
#define __ASM_SMALL_VALUE 0x01000000
#define __ASM_NO_TYPE 0x02000000
#define __ASM_NO_LOCAL 0x04000000
#if __RCX==2
// no restriction
```

```
#define __ASM_SRC_BASIC 0
#define __ASM_SRC_EXT 0
#else
#define __ASM_SRC_BASIC 0x000005
#define __ASM_SRC_EXT 0x000015
#endif
```

El flag __ASM_SMALL_VALUE indica que el valor de un byte debe ser usado en vez de un valor de dos bytes. El flag __ASM_NO_TYPE indica que el código fuente debe ser omitido. El flag __ASM_NO_LOCAL especifica que las variables locales no son una fuente legal para la expresión. Debe tenerse en cuenta que el firmware de RCX2 es menos restrictivo que los otros intérpretes, por lo que la definición de __ASM_SRC_BASIC y __ASM_SRC_EXT son menos estrictas en el caso del RCX2. El archivo de definición de API para NQC contiene numerosos ejemplos del uso de restrictores dentro de una instrucción ASM. Si está usando una versión de línea de comandos de NQC, puede editar el archivo API tecleando el siguiente comando:

```
nqc –api
```

4.2. Fuentes de datos

Los intérpretes del bytecode usan diferentes fuentes de datos para representar diferentes tipos de datos (constantes, variables, números aleatorios, valores de sensores, etc). Las fuentes específicas dependen de que tipo de dispositivo se esté usando y están descritas en la documentación del SDK de LEGO.

NQC ofrece un operador especial para representar una fuente de datos:

```
@ constant
```

El valor de esta expresión es la fuente de datos descrita por la constante. Los 16 bits de la constante representan el valor de los datos, y los siguientes 8 bits son el código fuente. Por ejemplo, el código fuente para un número aleatorio es 4, por lo tanto la expresión para un número aleatorio entre 0 y 9 sería:

```
@ 0x40009
```

El archivo API de NQC define un número de macros que usan el operador @ transparente para el programador. Por ejemplo, en el caso de números aleatorios:

```
# defineRandom(n) @(0x40000 + (n))
```

Adviértase que desde que la fuente 0 es un espacio de variable global, las posiciones de memoria global pueden ser referenciadas de modo numérico: @0 se refiere a la posición 0. Si por alguna razón necesita tener un control sobre dónde se almacenan las variables, entonces debe usar #pragma reserve para ordenar a NQC que no use éstas, y entonces acceder manualmente con el operador @. Por ejemplo, el siguiente código reserva el espacio 0 y crea una macro para llamarla x:

```
#pragma reserve 0
#define x (@0)
```